This page explains how to use container-native load balancing in Google Kubernetes Engine.

Container-native load balancing enables HTTP(S) load balancers (/load-balancing/docs) to target Pods directly and to evenly distribute their traffic to Pods.

Container-native load balancing leverages a data model called *network endpoint groups* (NEGs) (/load-balancing/docs/negs), collections of network endpoints represented by IP-port pairs.

Container-native load balancing offers the following benefits:

Pods are first-class citizens for load balancing

kube-proxy (https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/) configures nodes' `iptables` rules to distribute traffic to Pods. Without container-native load balancing, load balancer traffic travels to the node instance groups and gets routed via `iptables` rules to Pods which might or might not be in the same node. With container-native load balancing, load balancer traffic is distributed directly to the Pods which should receive the traffic, eliminating the extra network hop. Container-native load balancing also helps with improved health checking since it targets Pods directly.

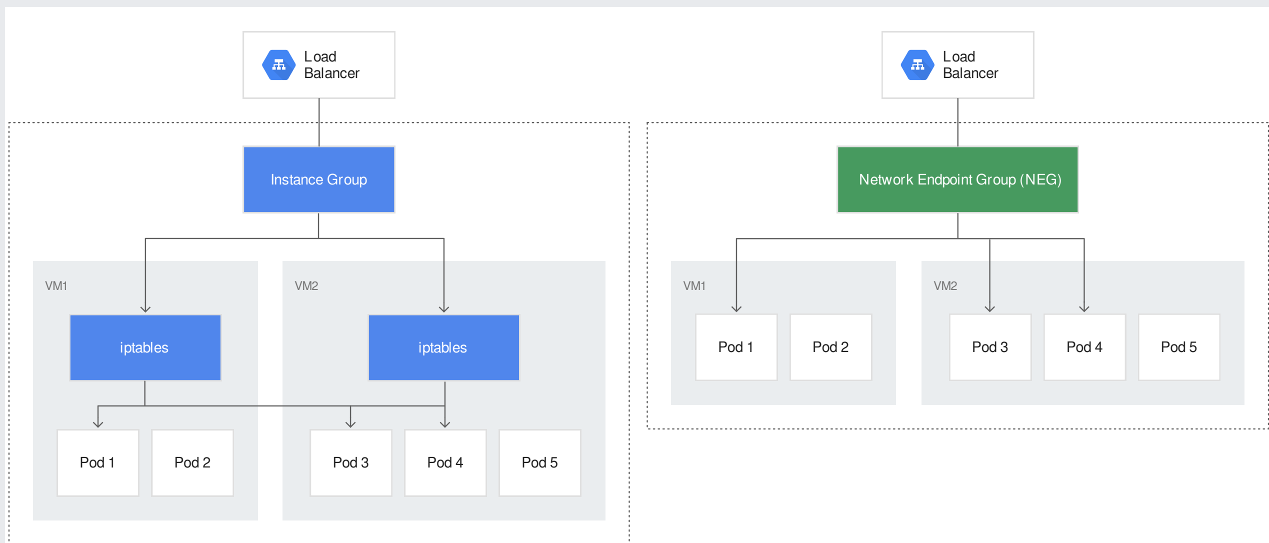

Diagram comparing default behavior (left) with container-native load balancer behavior.

Improved network performance

Because the container-native load balancer talks directly with the Pods, and connections have fewer network hops, both latency and throughput are improved.

Increased visibility

With container-native load balancing, you have visibility into the round-trip time (RTT) from the client to the HTTP(S) load balancer, including Stackdriver UI support. This makes troubleshooting your services at the NEG-level easier.

Support for HTTP(S) Load Balancing features

Container-native load balancing offer native support in Google Kubernetes Engine for several features of HTTP(S) Load Balancing, such as integration with GCP services like Google Cloud Armor (/kubernetes-engine/docs/how-to/cloud-armor-backendconfig), Cloud CDN (/kubernetes-engine/docs/how-to/cdn-backendconfig), and Identity-Aware Proxy (/iap/docs/enabling-kubernetes-howto). It also features load balancing algorithms for accurate traffic distribution.

Support for Traffic Director

The NEG data model is required to use Traffic Director (/traffic-director/docs), Google Cloud's fully managed traffic control plane for service mesh.

For relevant Pods, the corresponding Ingress controller manages a readiness gate (https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#pod-readiness-gate) of type `cloud.google.com/load-balancer-neg-ready`. The Ingress controller polls the load balancer's health check status (/load-balancing/docs/health-check-concepts), which includes the health of all endpoints in the NEG. When the load balancer's health check status indicates that the endpoint corresponding to a particular Pod is healthy, the Ingress controller sets the Pod's readiness gate value to `True`. The kubelet running on each Node then computes the Pod's effective readiness, considering both the value of this readiness gate and, if defined, the Pod's readiness probe

(https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/#define-readiness-probes)

.

For container-native load balancing, Pod readiness gates are automatically enabled in:

- v1.13 GKE clusters running v1.13.8 and higher
- v1.14 GKE clusters running v1.14.4 and higher

Readiness gates control the rate of a rolling update. GKE versions listed above automatically add readiness gates to Pods. When you initiate a rolling update, as GKE creates new Pods, an endpoint for each new Pod is added to a NEG. When the endpoint is healthy from the perspective of the load balancer, the Ingress controller sets the readiness gate to True. Thus, a newly created Pod must at least pass its readiness gate *before* GKE removes an old Pod. This ensures that the corresponding endpoint for the Pod has already passed the load balancer's health check and that the backend capacity is maintained.

If a Pod's readiness gate never indicates that the Pod is ready, due to a bad container image or a misconfigured load balancer health check, the load balancer won't direct traffic to the new Pod. If such a failure occurs while rolling out an updated Deployment, the rollout stalls after attempting to create one new Pod because that Pod's readiness gate is never True. See the troubleshooting section (/kubernetes-engine/docs/how-to/container-native-load-balancing#stalled_rollout) for information on how to detect and fix this situation.

Without container-native load balancing and readiness gates, GKE can't detect if a load balancer's endpoints are healthy before marking Pods as ready. In previous Kubernetes versions, you control the rate that Pods are removed and replaced by specifying a delay period (`minReadySeconds` in the Deployment specification).

Container-native load balancers on Google Kubernetes Engine have the following requirements:

Google Kubernetes Engine v1.13.8 or v1.14.4

Container-native load balancers are generally available in:

- v1.13 GKE clusters running v1.13.8 and higher
- v1.14 GKE clusters running v1.14.4 and higher

VPC-native

To use container-native load balancing, clusters must be VPC-native. To learn more, refer to Creating VPC-native clusters using Alias IPs (/kubernetes-engine/docs/how-to/alias-ips).

HTTP load balancing

To use container-native load balancing, your cluster must have HTTP load-balancing enabled. GKE clusters have HTTP load-balancing enabled by default; you must not disable it.

Container-native load balancers do not work with legacy networks (/vpc/docs/legacy).

Container-native load balancers do not support internal load balancers or network load balancers.

You are charged for the HTTP(S) load balancer provisioned by the Ingress that you create in this guide. For load balancer pricing information, refer to Load balancing and forwarding rules (/compute/pricing#lb) on the Compute Engine pricing page.

The following sections walk you through a container-native load balancing configuration on Google Kubernetes Engine.

To use container-native load balancing, you must create a cluster with alias IPs (/kubernetes-engine/docs/how-to/alias-ips#procedures) enabled.

For example, the following command creates a cluster, `neg-demo-cluster`, with an auto-provisioned subnetwork in zone `us-central1-a`:

Next, deploy a workload to the cluster.

The following sample Deployment (/kubernetes-engine/docs/concepts/deployment), `neg-demo-app`, runs a single instance of a containerized HTTP server. We recommend you use workloads that use Pod Readiness feedback. See the Pod readiness section (#pod_readiness) above for more information and for GKE version requirements.

In this Deployment, each container runs an HTTP server. The HTTP server simply returns the hostname of the application server (the name of the Pod on which the server runs) as a response.

Save this manifest as `neg-demo-app.yaml`, then create the Deployment by running the following command:

After you have created a Deployment, you need to group its Pods into a Service (https://kubernetes.io/docs/concepts/services-networking/service/).

The following sample Service, `neg-demo-svc`, targets the sample Deployment you created in the previous section:

The Service's annotation, `cloud.google.com/neg: '{"ingress": true}'`, enables container-native load balancing. However, the load balancer is not created until you create an Ingress (#create_ingress) for the Service.

Save this manifest as `neg-demo-svc.yaml`, then create the Service by running the following command:

The following sample Ingress (https://kubernetes.io/docs/concepts/services-networking/ingress/), `neg-demo-ing`, targets the Service you created:

Save this manifest as `neg-demo-ing.yaml`, then create the Ingress by running the following command:

Upon creating the Ingress, an HTTP(S) load balancer is created in the project, and NEGs are created in each zone in which the cluster runs. The endpoints in the NEG and the endpoints of the Service are kept in sync.

After you have deployed a workload, grouped its Pods into a Service, and created an Ingress for the Service, you should verify that the Ingress has provisioned the container-native load balancer successfully.

To retrieve the status of the Ingress, run the following command:

In the command output, look for `ADD` and `CREATE` events:

The following sections explain how you can test the functionality of a container-native load balancer.

Wait several minutes for the HTTP(S) load balancer to be configured.

You can verify that the container-native load balancer is functioning by visiting the Ingress' IP address.

To get the Ingress IP address, run the following command:

In the command output, the Ingress' IP address is displayed in the `ADDRESS` column. Visit the IP address in a web browser.

You can also get the health status of the load balancer's backend service (/load-balancing/docs/backend-service).

First, get a list of the backend services running in your project:

Copy of the name of the backend that includes the name of the Service, such as `neg-demo-svc`. Then, get the health status of the backend service:

Another way you can test that the load balancer functions as expected is by scaling the sample Deployment, sending test requests to the Ingress, and verifying that the correct number of replicas respond.

The following command scales the `neg-demo-app` Deployment from one instance to two instances:

Wait a few minutes for the rollout to complete. To verify that the rollout is complete, run the following command:

In the command output, verify that there are two available replicas:

Then, run the following command to count the number of distinct responses from the load balancer:

where ***[IP_ADDRESS]*** is the Ingress' IP address. You can get the Ingress' IP address from `kubectl describe ingress neg-demo-ing`.

If this command returns a 404 error, wait a few minutes for the load balancer to come up, then try again.

In the command output, observe that the number of distinct responses is the same as the number of replicas, indicating that all backend Pods are serving traffic:

After completing the tasks on this page, follow these steps to remove the resources to prevent unwanted charges incurring on your account:

Use the techniques below to verify your networking configuration. The following sections explain how to resolve specific issues related to container-native load balancing.

- See the load balancing documentation (/load-balancing/docs/negs/setting-up-negs#listing-negs) for how to list your network endpoint groups.
- You can find the name and zones of the NEG that corresponds to a service in the `neg-status` annotation of the service. Get the Service specification with:

  The `metadata:annotations:cloud.google.com/neg-status` annotation lists the name of service's corresponding NEG and the zones of the NEG.
- You can check the health of the backend service that corresponds to a NEG with the following command:

The backend service has the same name as its NEG.

- To print a service's event logs:

The service's name string includes the name and namespace of the corresponding GKE Service.

Symptoms

When you attempt to create a cluster with alias IPs, you might encounter the following error:

Potential causes

You encounter this error if you attempt to create a cluster with alias IPs that also uses a legacy network.

Resolution

Ensure that you do not create a cluster with alias IPs and a legacy network enabled simultaneously. For more information about using alias IPs, refer to Creating VPC-native clusters using Alias IPs (/kubernetes-engine/docs/how-to/alias-ips).

Symptoms

502 errors or rejected connections.

Potential causes

New endpoints generally become reachable after attaching them to the load balancer, provided that they respond to health checks. You might encounter 502 errors or rejected connections if traffic cannot reach the endpoints.

502 errors and rejected connections can also be caused by a container that doesn't handle `SIGTERM`. If a container doesn't explicitly handle `SIGTERM`, it immediately terminates and stops handling requests. The load balancer continues to send incoming traffic to the terminated container, leading to errors.

Resolution

Configure containers to handle `SIGTERM` and continue responding to requests throughout the termination grace period (30 seconds by default). Configure Pods to begin failing health checks when they receive `SIGTERM`. This signals the load balancer to stop sending traffic to the Pod while endpoint deprograming is in progress.

See the documentation on Pod termination (https://kubernetes.io/docs/concepts/workloads/pods/pod/#termination-of-pods) and this post about Pod termination best practices (https://cloud.google.com/blog/products/gcp/kubernetes-best-practices-terminating-with-grace) for more information.

To troubleshoot traffic not reaching the endpoints, verify that firewall rules allow incoming TCP traffic to your endpoints in the `130.211.0.0/22` and `35.191.0.0/16` ranges. To learn more, refer to Adding Health Checks (/load-balancing/docs/health-checks) in the Cloud Load Balancing documentation.

View the backend services in your project. The name string of the relevant backend service includes the name and namespace of the corresponding Google Kubernetes Engine Service:

Retrieve the backend health status from the backend service:

If all backends are unhealthy, your firewall, Ingress, or Service might be misconfigured.

If some backends are unhealthy for a short period of time, network programming latency might be the cause.

If some backends do not appear in the list of backend services, programming latency might be the cause. You can verify this by running the following command, where ***[NEG]*** is the name of the backend service. (NEGs and backend services share the same name):

Check if all the expected endpoints are in the NEG.

Symptoms

Rolling out an updated Deployment stalls, and the number of up-to-date replicas does not match the desired number of replicas.

Potential causes

The deployment's health checks are failing. The container image might be bad or the health check might be misconfigured. The rolling replacement of Pods waits until the newly started Pod passes its Pod readiness gate (https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#pod-readiness-gate). This only occurs if the Pod is responding to load balancer health checks. If the Pod does not respond, or if the health check is misconfigured, the readiness gate conditions can't be met and the rollout can't continue.

If you're using `kubectl` 1.13 or higher, you can check the status of a Pod's readiness gates with the following command:

Check the `READINESS GATES` column.

This column doesn't exist in `kubectl` 1.12 and lower. A Pod that is marked as being in the `READY` state may have a failed readiness gate. To verify this, use the following command:

The readiness gates and their status are listed in the output.

Resolution

Verify that the container image in your Deployment's Pod specification is functioning correctly and is able to respond to health checks. Verify that the health checks are correctly configured.

Container-native load balancing on Google Kubernetes Engine has the following known issues:

Google Kubernetes Engine garbage collects container-native load balancers every two minutes. If a cluster is deleted before load balancers are fully removed, you need to manually delete the load balancer's NEGs.

View the NEGs in your project by running the following command:

In the command output, look for the relevant NEGs.

To delete a NEG, run the following command, where ***[NEG]*** is the name of the NEG:

This issue does not occur in clusters that use Pod readiness feedback to manage workload rollouts. See the Pod reac
n (#pod_readiness) for more information.

When you deploy a workload to your cluster, or when you update an existing workload, the container-native load balancer can take longer to propagate new endpoints than it takes to finish the workload rollout. The sample Deployment that you deploy in this guide uses two fields to align its rollout with the propagation of endpoints: `terminationGracePeriodSeconds` and `minReadySeconds`.

`terminationGracePeriodSeconds` (https://kubernetes.io/docs/concepts/workloads/pods/pod/#termination-of-pods) allows the Pod to shut down gracefully by waiting for connections to terminate after a Pod is scheduled for deletion.

`minReadySeconds` (https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#min-ready-seconds) adds a latency period after a Pod is created. You specify a minimum number of seconds for which a new Pod should be in Ready status (https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#pod-conditions), without any of its containers crashing, for the Pod to be considered available.

You should configure your workloads' `minReadySeconds` and `terminationGracePeriodSeconds` values to be 60 seconds or higher to ensure that the service is not disrupted due to workload rollouts.

`terminationGracePeriodSeconds` is available in all Pod specifications, and `minReadySeconds` is available for Deployments and DaemonSets.

To learn more about fine-tuning rollouts, refer to RollingUpdateStrategy (https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#strategy).

- Learn more about NEGs (/load-balancing/docs/negs).
- Learn more about VPC-native clusters (/kubernetes-engine/docs/how-to/alias-ips).
- Learn more about HTTP(S) Load Balancing (/load-balancing/docs).
- Watch a KubeCon talk about Pod readiness gates (https://www.youtube.com/watch?v=Vw9GmSeomFg).